Redex: Practical Semantics Engineering

Version 4.2.4

Robert Bruce Findler
and Casey Klein

January 28, 2010

PLT Redex consists of a domain-specific language for specifying reduction semantics, plus a suite of tools for working with the semantics.

This is a reference manual for Redex. See `http://redex.plt-scheme.org/` for a gentler overview. (See also the `examples` subdirectory in the `redex` collection.)

To load Redex use:

```
(require redex)
```

which provides all of the names documented in this library.

The module `redex/reduction-semantics` provides only the non-GUI portions of what is described in this manual (everything except the last two sections), making it suitable for use with `mzscheme` scripts.

Contents

1 Patterns

```
(require redex/reduction-semantics)
```

All of the exports in this section are provided both by `redex/reduction-semantics` (which includes all non-GUI portions of Redex) and also exported by `redex` (which includes all of Redex).

This section covers Redex's *pattern* language, used in many of Redex's forms.

Note that pattern matching is caching (including caching the results of side-conditions). This means that once a pattern has matched a given term, Redex assumes that it will always match that term.

This is the grammar for the Redex pattern language. Non-terminal references are wrapped with angle brackets; otherwise identifiers in the grammar are terminals.

```
         pattern = any
                 | number
                 | natural
                 | integer
                 | real
                 | string
                 | variable
                 | (variable-except <id> ...)
                 | (variable-prefix <id>)
                 | variable-not-otherwise-mentioned
                 | hole
                 | symbol
                 | (name <id> <pattern>)
                 | (in-hole <pattern> <pattern>)
                 | (hide-hole <pattern>)
                 | (side-condition <pattern> guard)
                 | (cross <id>)
                 | (<pattern-sequence> ...)
                 | <scheme-constant>

pattern-sequence = <pattern>
                 | ... ; literal ellipsis
                 | ..._id
```

- The `any` pattern matches any sexpression. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `number` pattern matches any number. This pattern may also be suffixed with an

underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.

- The `natural` pattern matches any exact non-negative integer. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `integer` pattern matches any exact integer. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `real` pattern matches any real number. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `string` pattern matches any string. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `variable` pattern matches any symbol. This pattern may also be suffixed with an underscore and another identifier, in which case they bind the full name (as if it were an implicit name pattern) and match the portion before the underscore.
- The `variable-except` pattern matches any symbol except those listed in its argument. This is useful for ensuring that keywords in the language are not accidentally captured by variables.
- The `variable-prefix` pattern matches any symbol that begins with the given prefix.
- The `variable-not-otherwise-mentioned` pattern matches any symbol except those that are used as literals elsewhere in the language.
- The `hole` pattern matches anything when inside the first argument to an in-hole pattern. Otherwise, it matches only a hole.
- The `symbol` pattern stands for a literal symbol that must match exactly, unless it is the name of a non-terminal in a relevant language or contains an underscore.

  If it is a non-terminal, it matches any of the right-hand sides of that non-terminal. If the non-terminal appears twice in a single pattern, then the match is constrained to expressions that are the same, unless the pattern is part of a grammar, in which case there is no constraint.

  If the symbol is a non-terminal followed by an underscore, for example `e_1`, it is implicitly the same as a name pattern that matches only the non-terminal, `(name e_1 e)` for the example. Accordingly, repeated uses of the same name are constrainted to match the same expression.

  If the symbol is a non-terminal followed by `_!_`, for example `e_!_1`, it is also treated as a pattern, but repeated uses of the same pattern are constrained to be different. For example, this pattern:

```
(e_!_1 e_!_1 e_!_1)
```

matches lists of three `e`s, but where all three of them are distinct.

Unlike a `_` pattern, the `_!_` patterns do not bind names.

If `_` names and `_!_` are mixed, they are treated as separate. That is, this pattern `(e_1 e_!_1)` matches just the same things as `(e e)`, but the second doesn't bind any variables.

If the symbol otherwise has an underscore, it is an error.

- The pattern `(name symbol pattern)` matches *pattern* and binds using it to the name `symbol`.
- The `(in-hole pattern pattern)` pattern matches the first *pattern*. This match must include exactly one match against the second *pattern*. If there are zero matches or more than one match, an exception is raised.

  When matching the first argument of in-hole, the `hole` pattern matches any sexpression. Then, the sexpression that matched the hole pattern is used to match against the second pattern.
- The `(hide-hole pattern)` pattern matches what the embedded *pattern* matches but if the pattern matcher is looking for a decomposition, it ignores any holes found in that *pattern*.
- The `(side-condition pattern guard)` pattern matches what the embedded *pattern* matches, and then the guard expression is evaluated. If it returns `#f`, the pattern fails to match, and if it returns anything else, the pattern matches. Any occurrences of `name` in the pattern (including those implicitly there via `_` pattersn) are bound using `term-let` in the guard.
- The `(cross symbol)` pattern is used for the compatible closure functions. If the language contains a non-terminal with the same name as `symbol`, the pattern `(cross symbol)` matches the context that corresponds to the compatible closure of that non-terminal.
- The `(pattern-sequence ...)` pattern matches a sexpression list, where each pattern-sequence element matches an element of the list. In addition, if a list pattern contains an ellipsis, the ellipsis is not treated as a literal, instead it matches any number of duplications of the pattern that came before the ellipses (including 0). Furthermore, each `(name symbol pattern)` in the duplicated pattern binds a list of matches to `symbol`, instead of a single match. (A nested duplicated pattern creates a list of list matches, etc.) Ellipses may be placed anywhere inside the row of patterns, except in the first position or immediately after another ellipses.

  Multiple ellipses are allowed. For example, this pattern:

  ```
  ((name x a) ... (name y a) ...)
  ```

  matches this sexpression:

  ```
  (term (a a))
  ```

three different ways. One where the first `a` in the pattern matches nothing, and the second matches both of the occurrences of `a`, one where each named pattern matches a single `a` and one where the first matches both and the second matches nothing.

If the ellipses is named (ie, has an underscore and a name following it, like a variable may), the pattern matcher records the length of the list and ensures that any other occurrences of the same named ellipses must have the same length.

As an example, this pattern:

```
((name x a) ..._1 (name y a) ..._1)
```

only matches this sexpression:

```
(term (a a))
```

one way, with each named pattern matching a single a. Unlike the above, the two patterns with mismatched lengths is ruled out, due to the underscores following the ellipses.

Also, like underscore patterns above, if an underscore pattern begins with `..._!_`, then the lengths must be different.

Thus, with the pattern:

```
((name x a) ..._!_1 (name y a) ..._!_1)
```

and the expression

```
(term (a a))
```

two matches occur, one where `x` is bound to `'()` and `y` is bound to `'(a a)` and one where `x` is bound to `'(a a)` and `y` is bound to `'()`.

```
(redex-match lang pattern any)
(redex-match lang pattern)
```

If `redex-match` receives three arguments, it matches the pattern (in the language) against its third argument. If it matches, this returns a list of match structures describing the matches. If it fails, it returns `#f`.

If `redex-match` receives only two arguments, it builds a procedure for efficiently testing if expressions match the pattern, using the language *`lang`*. The procedures accepts a single expression and if the expresion matches, it returns a list of match structures describing the matches. If the match fails, the procedure returns `#f`.

```
(match? val) → boolean?
  val : any/c
```

Determines if a value is a `match` structure.

```
(match-bindings m) → (listof bind?)
```

```
m : match?
```

This returns a bindings structure (see below) that binds the pattern variables in this match.

```
(struct bind (name exp))
  name : symbol?
  exp : any/c
```

Instances of this struct are returned by `redex-match`. Each `bind` associates a name with an s-expression from the language, or a list of such s-expressions, if the `(name ...)` clause is followed by an ellipsis. Nested ellipses produce nested lists.

```
(caching-enabled?) → boolean?
(caching-enabled? on?) → void?
  on? : boolean?
```

When this parameter is `#t` (the default), Redex caches the results of pattern matching and metafunction evaluation. There is a separate cache for each pattern and metafunction; when one fills (see `set-cache-size!`), Redex evicts all of the entries in that cache.

Caching should be disabled when matching a pattern that depends on values other than the in-scope pattern variables or evaluating a metafunction that reads or writes mutable external state.

```
(set-cache-size! size) → void?
  size : positive-integer?
```

Changes the size of the per-pattern and per-metafunction caches. The default size is `350`.

2 Terms

All of the exports in this section are provided both by `redex/reduction-semantics` (which includes all non-GUI portions of Redex) and also exported by `redex` (which includes all of Redex).

Object langauge expressions in Redex are written using `term`. It is similar to Scheme's `quote` (in many cases it is identical) in that it constructs lists as the visible representation of terms.

The grammar of *terms* is (note that an ellipsis stands for repetition unless otherwise indicated):

```
         term = identifier
              | (term-sequence ...)
              | ,scheme-expression
              | (in-hole term term)
              | hole
              | #t
              | #f
              | string

term-sequence = term
              | ,@scheme-expression
              | ... ; literal ellipsis
```

- A term written *`identifier`* is equivalent to the corresponding symbol, unless the identifier is bound by `term-let` (or in a pattern elsewhere) or is `hole` (as below).
- A term written `(term-sequence ...)` constructs a list of the terms constructed by the sequence elements.
- A term written `,scheme-expression` evaluates the `scheme-expression` and substitutes its value into the term at that point.
- A term written `,@scheme-expression` evaluates the `scheme-expression`, which must produce a list. It then splices the contents of the list into the expression at that point in the sequence.
- A term written `(in-hole tttterm tttterm)` is the dual to the pattern `in-hole` – it accepts a context and an expression and uses `plug` to combine them.
- A term written `hole` produces a hole.
- A term written as a literal boolean or a string produces the boolean or the string.

```
(term term)
```

This form is used for construction of a term.

It behaves similarly to `quasiquote`, except for a few special forms that are recognized (listed below) and that names bound by `term-let` are implicitly substituted with the values that those names were bound to, expanding ellipses as in-place sublists (in the same manner as syntax-case patterns).

For example,

```
(term-let ([body '(+ x 1)]
           [(expr ...) '(+ - (values * /))]
           [((id ...) ...) '((a) (b) (c d))])
  (term (let-values ([(id ...) expr] ...) body)))
```

evaluates to

```
'(let-values ([(a) +]
              [(b) -]
              [(c d) (values * /)])
   (+ x 1))
```

It is an error for a term variable to appear in an expression with an ellipsis-depth different from the depth with which it was bound by `term-let`. It is also an error for two `term-let`-bound identifiers bound to lists of different lengths to appear together inside an ellipsis.

```
hole
```

Recognized specially within `term`. A `hole` form is an error elsewhere.

```
in-hole
```

Recognized specially within `reduction-relation`. An `in-hole` form is an error elsewhere.

```
(term-let ([tl-pat expr] ...) body)

    tl-pat = identifier
           | (tl-pat-ele ...)

tl-pat-ele = tl-pat
           | tl-pat ... ; a literal ellipsis
```

Matches each given id pattern to the value yielded by evaluating the corresponding expr and binds each variable in the id pattern to the appropriate value (described below). These bindings are then accessible to the *term* syntactic form.

Note that each ellipsis should be the literal symbol consisting of three dots (and the ... elsewhere indicates repetition as usual). If *tl-pat* is an identifier, it matches any value and binds it to the identifier, for use inside `term`. If it is a list, it matches only if the value being matched is a list value and only if every subpattern recursively matches the corresponding list element. There may be a single ellipsis in any list pattern; if one is present, the pattern before the ellipses may match multiple adjacent elements in the list value (possibly none).

This form is a lower-level form in Redex, and not really designed to be used directly. If you want a `let`-like form that uses Redex's full pattern matching facilities, see `term-match` and `term-match/single`.

`(term-match language [pattern expression] ...)`

This produces a procedure that accepts term (or quoted) expressions and checks them against each pattern. The function returns a list of the values of the expression where the pattern matches. If one of the patterns matches multiple times, the expression is evaluated multiple times, once with the bindings in the pattern for each match.

When evaluating a `term-match` expression, the patterns are compiled in an effort to speed up matching. Using the procedural result multiple times to avoid compiling the patterns multiple times.

`(term-match/single language [pattern expression] ...)`

This produces a procedure that accepts term (or quoted) expressions and checks them against each pattern. The function returns the expression behind the first sucessful match. If that pattern produces multiple matches, an error is signaled. If no patterns match, an error is signaled.

Raises an exception recognized by `exn:fail:redex?` if no clauses match or if one of the clauses matches multiple ways.

When evaluating a `term-match/single` expression, the patterns are compiled in an effort to speed up matching. Using the procedural result multiple times to avoid compiling the patterns multiple times.

```
(plug context expression) → any
  context : any/c
  expression : any/c
```

The first argument to this function is an sexpression to plug into. The second argument is the sexpression to replace in the first argument. It returns the replaced term. This is also used when a `term` sub-expression contains `in-hole`.

```
(variable-not-in t var) → symbol?
  t : any/c
  var : symbol?
```

This helper function accepts an sexpression and a variable. It returns a variable not in the sexpression with a prefix the same as the second argument.

```
(variables-not-in t vars) → (listof symbol?)
  t : any/c
  vars : (listof symbol?)
```

This function, like variable-not-in, makes variables that do no occur in its first argument, but it returns a list of such variables, one for each variable in its second argument.

Does not expect the input symbols to be distinct, but does produce variables that are always distinct.

```
(exn:fail:redex? v) → boolean?
  v : any/c
```

Returns `#t` if its argument is a Redex exception record, and `#f` otherwise.

3 Languages

All of the exports in this section are provided both by `redex/reduction-semantics` (which includes all non-GUI portions of Redex) and also exported by `redex` (which includes all of Redex).

```
(define-language lang-name
  (non-terminal-spec pattern ...)
  ...)

non-terminal-spec = symbol
                  | (symbol ...)
```

This form defines the grammar of a language. It allows the definition of recursive patterns, much like a BNF, but for regular-tree grammars. It goes beyond their expressive power, however, because repeated `name` patterns and side-conditions can restrict matches in a context-sensitive way.

The non-terminal-spec can either by a symbol, indicating a single name for this non-terminal, or a sequence of symbols, indicating that all of the symbols refer to these productions.

As a simple example of a grammar, this is the lambda calculus:

```
(define-language lc-lang
  (e (e e ...)
     x
     v)
  (c (v ... c e ...)
     hole)
  (v (lambda (x ...) e))
  (x variable-not-otherwise-mentioned))
```

with non-terminals `e` for the expression language, `x` for variables, `c` for the evaluation contexts and `v` for values.

```
(define-extended-language language language
  (non-terminal pattern ...)
  ...)
```

This form extends a language with some new, replaced, or extended non-terminals. For example, this language:

```
(define-extended-language lc-num-lang
  lc-lang
  (v ....     ; extend the previous 'v' non-terminal
```

```
        +
        number)
    (x (variable-except lambda +)))
```

extends lc-lang with two new alternatives for the `v` non-terminal, carries forward the `e` and `c` non-terminals, and replaces the `x` non-terminal with a new one (which happens to be equivalent to the one that would have been inherited).

The four-period ellipses indicates that the new language's non-terminal has all of the alternatives from the original language's non-terminal, as well as any new ones. If a non-terminal occurs in both the base language and the extension, the extension's non-terminal replaces the originals. If a non-terminal only occurs in either the base language, then it is carried forward into the extension. And, of course, extend-language lets you add new non-terminals to the language.

If a language is has a group of multiple non-terminals defined together, extending any one of those non-terminals extends all of them.

`(language-nts lang) → (listof symbol?)`
  `lang : compiled-lang?`

Returns the list of non-terminals (as symbols) that are defined by this language.

`(compiled-lang? l) → boolean?`
  `l : any/c`

Returns `#t` if its argument was produced by `language`, `#f` otherwise.

4 Reduction Relations

All of the exports in this section are provided both by `redex/reduction-semantics` (which includes all non-GUI portions of Redex) and also exported by `redex` (which includes all of Redex).

```
(reduction-relation language domain main-arrow reduction-case ...)

        domain =
               | #:domain pattern

    main-arrow =
               | #:arrow arrow

reduction-case = (-> pattern term extras ...)

        extras = name
               | (fresh fresh-clause ...)
               | (side-condition scheme-expression)
               | (where tl-pat term)

  fresh-clause = var
               | ((var1 ...) (var2 ...))

        tl-pat = identifier
               | (tl-pat-ele ...)

    tl-pat-ele = tl-pat
               | tl-pat ... ; a literal ellipsis
```

Defines a reduction relation casewise, one case for each of the clauses beginning with `->` (or with `arrow`, if specified). Each of the `pattern`s refers to the *`language`*, and binds variables in the *`term`*.

Following the pattern and term can be the name of the reduction rule, declarations of some fresh variables, and/or some side-conditions. The name can either be a literal name (identifier), or a literal string.

The fresh variables clause generates variables that do not occur in the term being matched. If the *`fresh-clause`* is a variable, that variable is used both as a binding in the rhs-exp and as the prefix for the freshly generated variable. (The variable does not have to be a non-terminal in the language of the reduction relation.)

The second case of a *`fresh-clause`* is used when you want to generate a sequence of variables. In that case, the ellipses are literal ellipses; that is, you must actually write ellipses in your rule. The variable *var1* is like the variable in first case of a *fresh-clause*, namely it is used to determine the prefix of the generated variables and it is bound in the right-hand side of the reduction rule, but unlike the single-variable fresh clause, it is bound to a sequence of variables. The variable *var2* is used to determine the number of variables generated and *var2* must be bound by the left-hand side of the rule.

The side-conditions are expected to all hold, and have the format of the second argument to the side-condition pattern, described above.

Each `where` clauses binds a variable and the side-conditions (and `where` clauses) that follow the where declaration are in scope of the where declaration. The bindings are the same as bindings in a `term-let` expression.

As an example, this

```
(reduction-relation
 lc-lang
 (--> (in-hole c_1 ((lambda (variable_i ...) e_body) v_i ...))
      (in-hole c_1 ,(foldl lc-subst
                           (term e_body)
                           (term (v_i ...))
                           (term (variable_i ...))))
      beta-v))
```

defines a reduction relation for the lambda-calculus above.

```
(reduction-relation
 language
 (arrow-var pattern term) ...
 with
 [(arrow pattern term)
  (arrow-var var var)] ...)
```

Defines a reduction relation with shortcuts. As above, the first section defines clauses of the reduction relation, but instead of using `-->`, those clauses can use any identifier for an arrow, as long as the identifier is bound after the `with` clause.

Each of the clauses after the `with` define new relations in terms of other definitions after the `with` clause or in terms of the main `-->` relation.

A `fresh` variable is always fresh with respect to the entire term, not just with respect to the part that matches the right-hand-side of the newly defined arrow.

As an example, this

```
(reduction-relation
 lc-num-lang
```

```
(==> ((lambda (variable_i ...) e_body) v_i ...)
     ,(foldl lc-subst
             (term e_body)
             (term (v_i ...))
             (term (variable_i ...))))
(==> (+ number_1 ...)
     ,(apply + (term (number_1 ...))))

with
[(-> (in-hole c_1 a) (in-hole c_1 b))
 (==> a b)])
```

defines reductions for the lambda calculus with numbers, where the ==> relation is defined by reducing in the context c.

```
(extend-reduction-relation reduction-relation language more ...)
```

This form extends the reduction relation in its first argument with the rules specified in *more*. They should have the same shape as the rules (including the `with` clause) in an ordinary *reduction-relation*.

If the original reduction-relation has a rule with the same name as one of the rules specified in the extension, the old rule is removed.

In addition to adding the rules specified to the existing relation, this form also reinterprets the rules in the original reduction, using the new language.

```
(union-reduction-relations r ...) → reduction-relation?
  r : reduction-relation?
```

Combines all of the argument reduction relations into a single reduction relation that steps when any of the arguments would have stepped.

```
(reduction-relation->rule-names r)
 → (listof (union false/c symbol?))
  r : reduction-relation?
```

Returns the names of all of the reduction relation's clauses (or false if there is no name for a given clause).

```
(compatible-closure reduction-relation lang non-terminal)
```

This accepts a reduction, a language, the name of a non-terminal in the language and returns the compatible closure of the reduction for the specified non-terminal.

```
(context-closure reduction-relation lang pattern)
```

This accepts a reduction, a language, a pattern representing a context (ie, that can be used as the first argument to `in-hole`; often just a non-terminal) in the language and returns the closure of the reduction in that context.

```
(reduction-relation? v) → boolean?
  v : any/c
```

Returns `#t` if its argument is a reduction-relation, and `#f` otherwise.

```
(apply-reduction-relation r t) → (listof any/c)
  r : reduction-relation?
  t : any/c
```

This accepts reduction relation, a term, and returns a list of terms that the term reduces to.

```
(apply-reduction-relation/tag-with-names r
                                          t)
 → (listof (list/c (union false/c string?) any/c))
  r : reduction-relation?
  t : any/c
```

Like `apply-reduction-relation`, but the result indicates the names of the reductions that were used.

```
(apply-reduction-relation* r t) → (listof any/c)
  r : reduction-relation?
  t : any/c
```

The function `apply-reduction-relation*` accepts a reduction relation and a term. Starting from *t*, it follows every reduction path and returns all of the terms that do not reduce further. If there are infinite reduction sequences that do not repeat, this function will not terminate (it does terminate if the only infinite reduction paths are cyclic).

```
->
```

Recognized specially within `reduction-relation`. A `->` form is an error elsewhere.

```
fresh
```

Recognized specially within `reduction-relation`. A `fresh` form is an error elsewhere.

`with`

Recognized specially within `reduction-relation`. A `with` form is an error elsewhere.

5 Metafunctions and Relations

All of the exports in this section are provided both by redex/reduction-semantics (which includes all non-GUI portions of Redex) and also exported by redex (which includes all of Redex).

```
(define-metafunction language
  contract
  [(name pattern ...) term extras ...]
  ...)

  contract =
           | id : pattern ... -> pattern

    extras = (side-condition scheme-expression)
           | (where tl-pat term)

    tl-pat = identifier
           | (tl-pat-ele ...)

tl-pat-ele = tl-pat
           | tl-pat ... ; a literal ellipsis
```

The define-metafunction form builds a function on sexpressions according to the pattern and right-hand-side expressions. The first argument indicates the language used to resolve non-terminals in the pattern expressions. Each of the rhs-expressions is implicitly wrapped in *term*.

If specified, the side-conditions are collected with and and used as guards on the case being matched. The argument to each side-condition should be a Scheme expression, and the pattern variables in the *pattern* are bound in that expression.

Raises an exception recognized by exn:fail:redex? if no clauses match, if one of the clauses matches multiple ways (and that leads to different results for the different matches), or if the contract is violated.

Note that metafunctions are assumed to always return the same results for the same inputs, and their results are cached, unless caching-enabled? is set to #f. Accordingly, if a metafunction is called with the same inputs twice, then its body is only evaluated a single time.

As an example, these metafunctions finds the free variables in an expression in the lc-lang above:

```
(define-metafunction lc-lang
```

```
  free-vars : e -> (x ...)
  [(free-vars (e_1 e_2 ...))
   (∪ (free-vars e_1) (free-vars e_2) ...)]
  [(free-vars x) (x)]
  [(free-vars (lambda (x ...) e))
   (- (free-vars e) (x ...))])
```

The first argument to define-metafunction is the grammar (defined above). Following that are three cases, one for each variation of expressions (e in lc-lang). The free variables of an application are the free variables of each of the subterms; the free variables of a variable is just the variable itself, and the free variables of a lambda expression are the free variables of the body, minus the bound parameters.

Here are the helper metafunctions used above.

```
(define-metafunction lc-lang
  ∪ : (x ...) ... -> (x ...)
  [(∪ (x_1 ...) (x_2 ...) (x_3 ...) ...)
   (∪ (x_1 ... x_2 ...) (x_3 ...) ...)]
  [(∪ (x_1 ...))
   (x_1 ...)]
  [(∪) ()])

(define-metafunction lc-lang
  - : (x ...) (x ...) -> (x ...)
  [(- (x ...) ()) (x ...)]
  [(- (x_1 ... x_2 x_3 ...) (x_2 x_4 ...))
   (- (x_1 ... x_3 ...) (x_2 x_4 ...))
   (side-condition (not (memq (term x_2) (term (x_3 ...)))))]
  [(- (x_1 ...) (x_2 x_3 ...))
   (- (x_1 ...) (x_3 ...))])
```

Note the side-condition in the second case of `-`. It ensures that there is a unique match for that case. Without it, `(term (- (x x) x))` would lead to an ambiguous match.

```
(define-metafunction/extension f language
  contract
  [(g pattern ...) term extras ...]
  ...)
```

Defines a metafunction *g* as an extension of an existing metafunction *f*. The metafunction *g* behaves as if *f*'s clauses were appended to its definition (with occurrences of *f* changed to *g* in the inherited clauses).

For example, `define-metafunction/extension` may be used to extend the free-vars function above to the forms introduced by the language lc-num-lang.

```
(define-metafunction/extension free-vars lc-num-lang
  free-vars-num : e -> (x ...)
  [(free-vars-num number)
   ()]
  [(free-vars-num (+ e_1 e_2))
   (∪ (free-vars-num e_1)
      (free-vars-num e_2))])
```

```
(in-domain? (metafunction-name term ...))
```

Returns `#t` if the inputs specified to *`metafunction-name`* are legtimate inputs according to *`metafunction-name`*'s contract, and `#f` otherwise.

```
(define-relation language
 [(name pattern ...) term ...] ...)

    tl-pat = identifier
           | (tl-pat-ele ...)

tl-pat-ele = tl-pat
           | tl-pat ... ; a literal ellipsis
```

The `define-relation` form builds a relation on sexpressions according to the pattern and right-hand-side expressions. The first argument indicates the language used to resolve non-terminals in the pattern expressions. Each of the rhs-expressions is implicitly wrapped in *`term`*.

Relations are like metafunctions in that they are called with arguments and return results (unlike in, say, prolog, where a relation definition would be able to synthesize some of the arguments based on the values of others).

Unlike metafunctions, relations check all possible ways to match each case, looking for a true result and if none of the clauses match, then the result is `#f`. If there are multiple expressions on the right-hand side of a relation, then all of them must be satisfied in order for that clause of the relation to be satisfied.

Note that relations are assumed to always return the same results for the same inputs, and their results are cached, unless `caching-enable?` is set to `#f`. Accordingly, if a relation is called with the same inputs twice, then its right-hand sides are evaluated only once.

```
(current-traced-metafunctions) → (or/c 'all (listof symbol?))
(current-traced-metafunctions traced-metafunctions) → void?
  traced-metafunctions : (or/c 'all (listof symbol?))
```

Controls which metafunctions are currently being traced. If it is `'all`, all of them are.

Otherwise, the elements of the list name the metafunctions to trace.

Defaults to `'()`.

6 Testing

All of the exports in this section are provided both by `redex/reduction-semantics` (which includes all non-GUI portions of Redex) and also exported by `redex` (which includes all of Redex).

```
(test-equal e1 e2)
```

Tests to see if *e1* is equal to *e2*.

```
(test-->> reduction-relation maybe-cycles e1 e2 ...)

cycles =
       | #:cycles-ok
```

Tests to see if the value of *e1* (which should be a term), reduces to the *e2*s under *reduction-relation* (using `apply-reduction-relation*`, so it may not terminate).

```
(test--> reduction-relation e1 e2 ...)
```

Tests to see if the value of *e1* (which should be a term), reduces to the *e2*s in a single step, under *reduction-relation* (using `apply-reduction-relation`).

```
(test-predicate p? e)
```

Tests to see if the value of *e* matches the predicate *p?*.

```
(test-results) → void?
```

Prints out how many tests passed and failed, and resets the counters so that next time this function is called, it prints the test results for the next round of tests.

```
(make-coverage subject)

subject = metafunction
        | relation-expr
```

Constructs a structure (recognized by `coverage?`) to contain per-case test coverage of the supplied metafunction or reduction relation. Use with `relation-coverage` and `covered-cases`.

```
(coverage? v) → boolean?
  v : any/c
```

Returns #t for a value produced by make-coverage and #f for any other.

```
(relation-coverage) → (listof coverage?)
(relation-coverage tracked) → void?
  tracked : (listof coverage?)
```

Redex populates the coverage records in *tracked* (default null), counting the times that tests exercise each case of the associated metafunction and relations.

```
(covered-cases c) → (listof (cons/c string? natural-number/c))
  c : coverage?
```

Extracts the coverage information recorded in *c*, producing an association list mapping names (or source locations, in the case of metafunctions or unnamed reduction-relation cases) to application counts.

Examples:

```
> (define-language empty-lang)
> (define-metafunction empty-lang
    [(plus number_1 number_2)
     ,(+ (term number_1) (term number_2))])
> (define equals
    (reduction-relation
     empty-lang
     (-> (+) 0 "zero")
     (-> (+ number) number)
     (-> (+ number_1 number_2 number ...)
          (+ (plus number_1 number_2)
             number ...)
          "add")))
> (let ([equals-coverage (make-coverage equals)]
        [plus-coverage (make-coverage plus)])
    (parameterize ([relation-coverage (list equals-coverage
                                             plus-coverage)])
      (apply-reduction-relation* equals (term (+ 1 2 3)))
      (values (covered-cases equals-coverage)
              (covered-cases plus-coverage))))
(("add" . 2) ("eval:4:0" . 1) ("zero" . 0))
(("eval:3:0" . 2))
```

```
(generate-term language pattern size-exp kw-args ...)

kw-args = #:attempts attempts-expr
        | #:retries retries-expr

  size-expr : natural-number/c
  attempt-num-expr : natural-number/c
  retries-expr : natural-number/c
```

Generates a random term matching `pattern` (in the given language).

The argument `size-expr` bounds the height of the generated term (measured as the height of the derivation tree used to produce the term).

The optional keyword argument `attempt-num-expr` (default `1`) provides coarse grained control over the random decisions made during generation. For example, the expected length of pattern-sequences increases with `attempt-num-expr`.

The random generation process does not actively consider the constraints imposed by side-condition or `_!_` patterns when constructing a term; instead, it tests the satisfaction of such constraints after it freely generates the relevant portion of the sub-term—regenerating the sub-term if necessary. The optional keyword argument *`retries-expr`* (default `100`) bounds the number of times that `generate-term` retries the generation of any sub-term. If `generate-term` is unable to produce a satisfying term after *`retries-expr`* attempts, it raises an exception recognized by `exn:fail:redex:generation-failure?`.

```
(redex-check language pattern property-expr kw-arg ...)

kw-arg = #:attempts attempts-expr
       | #:source metafunction
       | #:source relation-expr
       | #:retries retries-expr

  property-expr : any/c
  attempts-expr : natural-number/c
  relation-expr : reduction-relation?
  retries-expr : natural-number/c
```

Searches for a counterexample to *`property-expr`*, interpreted as a predicate universally quantified over the pattern variables bound by `pattern`. `redex-check` constructs and tests a candidate counterexample by choosing a random term *t* that matches `pattern` then evaluating *`property-expr`* using the `match-bindings` produced by `match`ing *t* against `pattern`.

redex-check generates at most *attempts-expr* (default 1000) random terms in its search. The size and complexity of terms it generates gradually increases with each failed attempt.

When passed a metafunction or reduction relation via the optional #:source argument, redex-check distributes its attempts across the left-hand sides of that metafunction/relation by using those patterns, rather than pattern, as the basis of its generation. It is an error if any left-hand side generates a term that does not match pattern.

Examples:

```
> (define-language empty-lang)
> (random-seed 0)
> (redex-check
   empty-lang
   ((number_1 ...)
    (number_2 ...))
   (equal? (reverse (append (term (number_1 ...))
                            (term (number_2 ...))))
           (append (reverse (term (number_1 ...)))
                   (reverse (term (number_2 ...))))))
redex-check: counterexample found after 2 attempts:
((1 0) (0))
> (redex-check
   empty-lang
   ((number_1 ...)
    (number_2 ...))
   (equal? (reverse (append (term (number_1 ...))
                            (term (number_2 ...))))
           (append (reverse (term (number_2 ...)))
                   (reverse (term (number_1 ...)))))
   #:attempts 200)
redex-check: no counterexamples in 200 attempts
> (let ([R (reduction-relation
            empty-lang
            (-> (Σ) 0)
            (-> (Σ number) number)
            (-> (Σ number_1 number_2 number_3 ...)
                (Σ ,(+ (term number_1) (term number_2))
                   number_3 ...)))])
    (redex-check
     empty-lang
     (Σ number ...)
     (printf "~s\n" (term (number ...)))
     #:attempts 3
     #:source R))
()
(0)
```

```
(1 0 0 3 0)
redex-check: no counterexamples in 1 attempt (with each clause)
```

```
(check-reduction-relation relation property kw-args ...)

kw-arg = #:attempts attempts-expr
       | #:retries retries-expr

  property : (-> any/c any/c)
  attempts-expr : natural-number/c
  retries-expr : natural-number/c
```

Tests *relation* as follows: for each case of *relation*, `check-reduction-relation` generates `attempts` random terms that match that case's left-hand side and applies *property* to each random term.

This form provides a more convenient notation for

```
(redex-check L any (property (term any))
             #:attempts (* n attempts)
             #:source relation)
```

when *relation* is a relation on `L` with `n` rules.

```
(check-metafunction metafunction property kw-args ...)

kw-arg = #:attempts attempts-expr
       | #:retries retries-expr

  property : (-> any/c any/c)
  attempts-expr : natural-number/c
  retries-expr : natural-number/c
```

Like `check-reduction-relation` but for metafunctions.

```
(exn:fail:redex:generation-failure? v) → boolean?
  v : any/c
```

Recognizes the exceptions raised by `generate-term`, `redex-check`, etc. when those forms are unable to produce a term matching some pattern.

Debugging PLT Redex Programs

It is easy to write grammars and reduction rules that are subtly wrong and typically such mistakes result in examples that just get stuck when viewed in a `traces` window.

The best way to debug such programs is to find an expression that looks like it should reduce but doesn't and try to find out what pattern is failing to match. To do so, use the `redex-match` special form, described above.

In particular, first ceck to see if the term matches the main non-terminal for your system (typically the expression or program nonterminal). If it does not, try to narrow down the expression to find which part of the term is failing to match and this will hopefully help you find the problem. If it does match, figure out which reduction rule should have matched, presumably by inspecting the term. Once you have that, extract a pattern from the left-hand side of the reduction rule and do the same procedure until you find a small example that shoudl work but doesn't (but this time you might also try simplifying the pattern as well as simplifying the expression).

7 GUI

```
(require redex/gui)
```

This section describes the GUI tools that Redex provides for exploring reduction sequences.

```
(traces  reductions
         expr
        [#:multiple? multiple?
         #:pred pred
         #:pp pp]
         #:colors colors
        [#:scheme-colors? scheme-colors?
         #:filter term-filter]
         #:x-spacing number?
         #:y-spacing number?
        [#:layout layout
         #:edge-labels? edge-label-font
         #:edge-label-font edge-label-font
         #:graph-pasteboard-mixin graph-pasteboard-mixin])
 → void?
  reductions : reduction-relation?
  expr : (or/c any/c (listof any/c))
  multiple? : boolean? = #f
  pred : (or/c (-> sexp any)                 = (lambda (x) #t)
               (-> sexp term-node? any))
  pp :  (or/c (any -> string)
              (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
  colors : (listof
            (cons/c string
                    (and/c (listof (or/c string? (is-a?/c color%)))
                           (lambda (x) (member (length x) '(2 3 4 6))))))
  scheme-colors? : boolean? = #t
  term-filter :  (-> any/c (or/c #f string?) any/c)
              = (lambda (x y) #t)
  number? : 15
  number? : 15
  layout : (-> (listof term-node?) void) = void
  edge-label-font : boolean? = #t
  edge-label-font : (or/c #f (is-a?/c font%)) = #f
  graph-pasteboard-mixin :  (make-mixin-contract graph-pasteboard<%>)
                         = values
```

This function opens a new window and inserts each expression in expr (if *multiple?* is #t – if *multiple?* is #f, then expr is treated as a single expression). Then, it reduces the terms until at least `reduction-steps-cutoff` (see below) different terms are found, or no more reductions can occur. It inserts each new term into the gui. Clicking the reduce button reduces until reduction-steps-cutoff more terms are found.

The *pred* function indicates if a term has a particular property. If it returns `#f`, the term is displayed with a pink background. If it returns a string or a `color%` object, the term is displayed with a background of that color (using `the-color-database` to map the string to a color). If it returns any other value, the term is displayed normally. If the pred function accepts two arguments, a term-node corresponding to the term is passed to the predicate. This lets the predicate function explore the (names of the) reductions that led to this term, using term-node-children, term-node-parents, and term-node-labels.

The *pred* function may be called more than once per node. In particular, it is called each time an edge is added to a node. The latest value returned determines the color.

The *pp* function is used to specially print expressions. It must either accept one or four arguments. If it accepts one argument, it will be passed each term and is expected to return a string to display the term.

If the *pp* function takes four arguments, it should render its first argument into the port (its second argument) with width at most given by the number (its third argument). The final argument is the text where the port is connected – characters written to the port go to the end of the editor.

The *colors* argument, if provided, specifies a list of reduction-name/color-list pairs. The traces gui will color arrows drawn because of the given reduction name with the given color instead of using the default color.

The `cdr` of each of the elements of *colors* is a list of colors, organized in pairs. The first two colors cover the colors of the line and the border around the arrow head, the first when the mouse is over a graph node that is connected to that arrow, and the second for when the mouse is not over that arrow. Similarly, the next colors are for the text drawn on the arrow and the last two are for the color that fills the arrow head. If fewer than six colors are specified, the colors specified colors are used and then defaults are filled in for the remaining colors.

The *scheme-colors?* argument, if `#t` causes `traces` to color the contents of each of the windows according to DrScheme's Scheme mode color Scheme. If it is `#f`, `traces` just uses black for the color scheme. In addition, Scheme-mode parenthesis highlighting is enabled when *scheme-colors?* is `#t` and not when it is `#f`.

The *term-filter* function is called each time a new node is about to be inserted into the graph. If the filter returns false, the node is not inserted into the graph.

The `x-spacing` and `y-spacing` control the amount of space put between the snips in the default layout.

The *layout* argument is called (with all of the terms) when new terms is inserted into the window. In general, it is called when after new terms are inserted in response to the user clicking on the reduce button, and after the initial set of terms is inserted. See also `term-node-set-position!`.

If `edge-labels?` is `#t` (the default), then edge labels are drawn; otherwise not.

The *edge-label-font* argument is used as the font on the edge labels. If `#f` is suppled, the `dc<%>` object's default font is used.

The traces library an instance of the `mrlib/graph` library's `graph-pasteboard<%>` interface to layout the graphs. Sometimes, overriding one of its methods can help give finer-grained control over the layout, so the *graph-pasteboard-mixin* is applied to the class before it is instantiated. Also note that all of the snips inserted into the editor by this library have a `get-term-node` method which returns the snip's `term-node`.

```
(traces/ps  reductions
            expr
            file
           [#:multiple? multiple?
            #:pred pred
            #:pp pp
            #:colors colors
            #:filter term-filter
            #:layout layout]
            #:x-spacing number?
            #:y-spacing number?
           [#:edge-labels? edge-label-font
            #:edge-label-font edge-label-font
            #:graph-pasteboard-mixin graph-pasteboard-mixin]
            #:post-process post-process)
 → void?
  reductions : reduction-relation?
  expr : (or/c any/c (listof any/c))
  file : (or/c path-string? path?)
  multiple? : boolean? = #f
  pred : (or/c (-> sexp any)              = (lambda (x) #t)
               (-> sexp term-node? any))
  pp :  (or/c (any -> string)
              (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
  colors : (listof (list string string)) = '()
  term-filter :  (-> any/c (or/c #f string?) any/c)
              = (lambda (x y) #t)
```

```
layout : (-> (listof term-node?) void) = void
number? : 15
number? : 15
edge-label-font : boolean? = #t
edge-label-font : (or/c #f (is-a?/c font%)) = #f
graph-pasteboard-mixin : (make-mixin-contract graph-pasteboard<%>)
                       = values
post-process : (-> (is-a?/c graph-pasteboard<%>) any/c)
```

This function behaves just like the function `traces`, but instead of opening a window to show the reduction graph, it just saves the reduction graph to the specified *file*.

All of the arguments behave like the arguments to `traces`, with the exception of the *post-process* argument. It is called just before the PostScript is created with the graph pasteboard.

```
(stepper reductions t [pp]) → void?
  reductions : reduction-relation?
  t : any/c
  pp : (or/c (any -> string)
             (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
```

This function opens a stepper window for exploring the behavior of its third argument in the reduction system described by its first two arguments.

The *pp* argument is the same as to the `traces` functions (above).

```
(stepper/seed reductions seed [pp]) → void?
  reductions : reduction-relation?
  seed : (cons/c any/c (listof any/c))
  pp : (or/c (any -> string)
             (any output-port number (is-a?/c text%) -> void))
     = default-pretty-printer
```

Like `stepper`, this function opens a stepper window, but it seeds it with the reduction-sequence supplied in *seed*.

```
(term-node-children tn) → (listof term-node?)
  tn : term-node?
```

Returns a list of the children (ie, terms that this term reduces to) of the given node.

Note that this function does not return all terms that this term reduces to – only those that are currently in the graph.

```
(term-node-parents tn) → (listof term-node?)
  tn : term-node?
```

Returns a list of the parents (ie, terms that reduced to the current term) of the given node.

Note that this function does not return all terms that reduce to this one – only those that are currently in the graph.

```
(term-node-labels tn) → (listof (or/c false/c string?))
  tn : term-node
```

Returns a list of the names of the reductions that led to the given node, in the same order as the result of term-node-parents. If the list contains `#f`, that means that the corresponding step does not have a label.

```
(term-node-set-color! tn color) → void?
  tn : term-node?
  color : (or/c string? (is-a?/c color%) false/c)
```

Changes the highlighting of the node; if its second argument is `#f`, the coloring is removed, otherwise the color is set to the specified `color%` object or the color named by the string. The `color-database<%>` is used to convert the string to a `color%` object.

```
(term-node-color tn) → (or/c string? (is-a?/c color%) false/c)
  tn : term-node?
```

Returns the current highlighting of the node. See also `term-node-set-color!`.

```
(term-node-set-red! tn red?) → void?
  tn : term-node?
  red? : boolean?
```

Changes the highlighting of the node; if its second argument is `#t`, the term is colored pink, if it is `#f`, the term is not colored specially.

```
(term-node-expr tn) → any
  tn : term-node?
```

Returns the expression in this node.

```
(term-node-set-position! tn x y) → void?
```

```
tn : term-node?
x : (and/c real? positive?)
y : (and/c real? positive?)
```

Sets the position of *tn* in the graph to (*x*,*y*).

```
(term-node-x tn) → real
  tn : term-node?
```

Returns the x coordinate of *tn* in the window.

```
(term-node-y tn) → real
  tn : term-node?
```

Returns the y coordinate of *tn* in the window.

```
(term-node-width tn) → real
  tn : term-node?
```

Returns the width of *tn* in the window.

```
(term-node-height tn) → real?
  tn : term-node?
```

Returns the height of *tn* in the window.

```
(term-node? v) → boolean?
  v : any/c
```

Recognizes term nodes.

```
(reduction-steps-cutoff) → number?
(reduction-steps-cutoff cutoff) → void?
  cutoff : number?
```

A parameter that controls how many steps the `traces` function takes before stopping.

```
(initial-font-size) → number?
(initial-font-size size) → void?
  size : number?
```

A parameter that controls the initial font size for the terms shown in the GUI window.

```
(initial-char-width) → (or/c number? (-> any/c number?))
(initial-char-width width) → void?
  width : (or/c number? (-> any/c number?))
```

A parameter that determines the initial width of the boxes where terms are displayed (measured in characters) for both the stepper and traces.

If its value is a number, then the number is used as the width for every term. If its value is a function, then the function is called with each term and the resulting number is used as the width.

```
(dark-pen-color) → (or/c string? (is-a?/c color<%>))
(dark-pen-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(dark-brush-color) → (or/c string? (is-a?/c color<%>))
(dark-brush-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-pen-color) → (or/c string? (is-a?/c color<%>))
(light-pen-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-brush-color) → (or/c string? (is-a?/c color<%>))
(light-brush-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(dark-text-color) → (or/c string? (is-a?/c color<%>))
(dark-text-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
(light-text-color) → (or/c string? (is-a?/c color<%>))
(light-text-color color) → void?
  color : (or/c string? (is-a?/c color<%>))
```

These six parameters control the color of the edges in the graph.

The dark colors are used when the mouse is over one of the nodes that is connected to this edge. The light colors are used when it isn't.

The pen colors control the color of the line. The brush colors control the color used to fill the arrowhead and the text colors control the color used to draw the label on the edge.

```
(default-pretty-printer v port width text) → void?
  v : any
  port : output-port
  width : number
  text : (is-a?/c text%)
```

This is the default value of pp used by traces and stepper and it uses pretty-print.

It sets the pretty-print-columns parameter to *width*, and it sets pretty-print-size-hook and pretty-print-print-hook to print holes and the symbol 'hole to match the way they are input in a term expression.

8 Typesetting

```
(require redex/pict)
```

The redex/pict library provides functions designed to automatically typeset grammars, reduction relations, and metafunction written with plt redex.

Each grammar, reduction relation, and metafunction can be saved in a .ps file (as encapsulated postscript), or can be turned into a pict for viewing in the REPL or using with Slideshow (see ***Slideshow**: PLT Figure and Presentation Tools*).

8.1 Picts & PostScript

This section documents two classes of operations, one for direct use of creating postscript figures for use in papers and for use in DrScheme to easily adjust the typesetting: render-term, render-language, render-reduction-relation, render-metafunctions, and render-lw, and one for use in combination with other libraries that operate on picts term->pict, language->pict, reduction-relation->pict, metafunction->pict, and lw->pict. The primary difference between these functions is that the former list sets dc-for-text-size and the latter does not.

```
(render-term lang term file) → (if file void? pict?)
  lang : compiled-lang?
  term : any/c
  file : (or/c #f path-string?)
```

Renders the term *term*. If *file* is #f, it produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename. See render-language for details on the construction of the pict.

```
(term->pict lang term) → pict?
  lang : compiled-lang?
  term : any/c
```

Produces a pict like render-term, but without adjusting dc-for-text-size.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-language lang [file #:nts nts]) → (if file void? pict?)
  lang : compiled-lang?
  file : (or/c false/c path-string?) = #f
```

```
  nts : (or/c false/c (listof (or/c string? symbol?)))
      = (render-language-nts)
```

Renders a language. If *file* is #f, it produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename. See render-language-nts for information on the *nts* argument.

This function parameterizes dc-for-text-size to install a relevant dc: a bitmap-dc% or a post-script-dc%, depending on whether *file* is a path.

See language->pict if you are using Slideshow or are otherwise setting dc-for-text-size.

```
(language->pict lang [#:nts nts]) → pict?
  lang : compiled-lang?
  nts : (or/c false/c (listof (or/c string? symbol?)))
      = (render-language-nts)
```

Produce a pict like render-language, but without adjusting dc-for-text-size.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-reduction-relation rel
                          [file
                           #:style style])
 → (if file void? pict?)
  rel : reduction-relation?
  file : (or/c false/c path-string?) = #f
  style : reduction-rule-style/c = (rule-pict-style)
```

Renders a reduction relation. If *file* is #f, it produces a pict; if *file* is a path, it saves Encapsulated PostScript in the provided filename. See rule-pict-style for information on the *style* argument.

This function parameterizes dc-for-text-size to install a relevant dc: a bitmap-dc% or a post-script-dc%, depending on whether *file* is a path. See also reduction-relation->pict.

The following forms of arrows can be typeset:

```
-> -+> ==> -> => ..> >-> ~~> ~> :-> :-> c-> ->>
 >- -< >>- -<<
```

```
(reduction-relation->pict r [#:style style]) → pict?
```

```
r : reduction-relation?
style : reduction-rule-style/c = (rule-pict-style)
```

Produces a pict like `render-reduction-relation`, but without setting `dc-for-text-size`.

This function is primarily designed to be used with Slideshow or with other tools that combine picts together.

```
(render-metafunction metafunction-name)
(render-metafunction metafunction-name filename)
(render-metafunctions metafunction-name ...)
(render-metafunctions metafunction-name ... #:file filename)
```

If provided with one argument, `render-metafunction` produces a pict that renders properly in the definitions window in DrScheme. If given two arguments, it writes postscript into the file named by *`filename`* (which may be either a string or bytes).

Similarly, `render-metafunctions` accepts multiple metafunctions and renders them together, lining up all of the clauses together.

This function sets `dc-for-text-size`. See also `metafunction->pict` and `metafunctions->pict`.

```
(metafunction->pict metafunction-name)
```

This produces a pict, but without setting `dc-for-text-size`. It is suitable for use in Slideshow or other libraries that combine picts.

```
(metafunctions->pict metafunction-name ...)
```

Like `metafunction->pict`, this produces a pict, but without setting `dc-for-text-size` and is suitable for use in Slideshow or other libraries that combine picts. Like `render-metafunctions`, it accepts multiple metafunctions and renders them together.

8.2 Customization

```
(render-language-nts) → (or/c false/c (listof symbol?))
(render-language-nts nts) → void?
  nts : (or/c false/c (listof symbol?))
```

The value of this parameter controls which non-terminals `render-language` and language->pict render by default. If it is #f (the default), all non-terminals are rendered. If it is a list of symbols, only the listed symbols are rendered.

See also language-nts.

```
(extend-language-show-union) → boolean?
(extend-language-show-union show?) → void?
  show? : boolean?
```

If this is #t, then a language constructed with extend-language is shown as if the language had been constructed directly with language. If it is #f, then only the last extension to the language is shown (with four-period ellipses, just like in the concrete syntax).

Defaultly #f.

Note that the #t variant can look a little bit strange if .... are used and the original version of the language has multi-line right-hand sides.

```
(render-reduction-relation-rules)
 → (or/c false/c (listof (or/c symbol? string?)))
(render-reduction-relation-rules rules) → void?
  rules : (or/c false/c (listof (or/c symbol? string?)))
```

This parameter controls which rules in a reduction relation will be rendered.

```
(rule-pict-style) → reduction-rule-style/c
(rule-pict-style style) → void?
  style : reduction-rule-style/c
```

This parameter controls the style used by default for the reduction relation. It can be 'horizontal, where the left and right-hand sides of the reduction rule are beside each other or 'vertical, where the left and right-hand sides of the reduction rule are above each other. The 'compact-vertical style moves the reduction arrow to the second line and uses less space between lines. The 'vertical-overlapping-side-conditions variant, the side-conditions don't contribute to the width of the pict, but are just overlaid on the second line of each rule. The 'horizontal-left-align style is like the 'horizontal style, but the left-hand sides of the rules are aligned on the left, instead of on the right.

```
reduction-rule-style/c : flat-contract?
```

A contract equivalent to

```
(symbols 'vertical
         'compact-vertical
```

```
            'vertical-overlapping-side-conditions
            'horizontal)
```

```
(arrow-space) → natural-number/c
(arrow-space space) → void?
  space : natural-number/c
```

This parameter controls the amount of extra horizontal space around the reduction relation arrow. Defaults to 0.

```
(horizontal-label-space) → natural-number/c
(horizontal-label-space space) → void?
  space : natural-number/c
```

This parameter controls the amount of extra space before the label on each rule, but only in horizontal mode. Defaults to 0.

```
(metafunction-pict-style)
 → (or/c 'left-right
          'up-down
          'left-right/vertical-side-conditions
          'up-down/vertical-side-conditions
          'left-right/compact-side-conditions
          'up-down/compact-side-conditions
          'left-right/beside-side-conditions)
(metafunction-pict-style style) → void?
  style : (or/c 'left-right
                'up-down
                'left-right/vertical-side-conditions
                'up-down/vertical-side-conditions
                'left-right/compact-side-conditions
                'up-down/compact-side-conditions
                'left-right/beside-side-conditions)
```

This parameter controls the style used for typesetting metafunctions. The `'left-right` style means that the results of calling the metafunction are displayed to the right of the arguments and the `'up-down` style means that the results are displayed below the arguments.

The `'left-right/vertical-side-conditions` and `'up-down/vertical-side-conditions` variants format side conditions each on a separate line, instead of all on the same line.

The `'left-right/compact-side-conditions` and `'up-down/compact-side-conditions` variants move side conditions to separate lines to avoid making the rendered form wider than it would be otherwise.

The 'left-right/beside-side-conditions variant is like 'left-right, except it puts the side-conditions on the same line, instead of on a new line below the case.

```
(metafunction-cases)
 → (or/c #f (and/c (listof (and/c integer?
                                   (or/c zero? positive?)))
                    pair?))
(metafunction-cases cases) → void?
  cases : (or/c #f (and/c (listof (and/c integer?
                                          (or/c zero? positive?)))
                           pair?))
```

This parameter controls which cases in a metafunction are rendered. If it is #f (the default), then all of the cases appear. If it is a list of numbers, then only the selected cases appear (counting from 0).

```
(label-style) → text-style/c
(label-style style) → void?
  style : text-style/c
(literal-style) → text-style/c
(literal-style style) → void?
  style : text-style/c
(metafunction-style) → text-style/c
(metafunction-style style) → void?
  style : text-style/c
(non-terminal-style) → text-style/c
(non-terminal-style style) → void?
  style : text-style/c
(non-terminal-subscript-style) → text-style/c
(non-terminal-subscript-style style) → void?
  style : text-style/c
(default-style) → text-style/c
(default-style style) → void?
  style : text-style/c
```

These parameters determine the font used for various text in the picts. See text in the texpict collection for documentation explaining text-style/c. One of the more useful things it can be is one of the symbols 'roman, 'swiss, or 'modern, which are a serif, sans-serif, and monospaced font, respectively. (It can also encode style information, too.)

The label-style is used for the reduction rule label names. The literal-style is used for names that aren't non-terminals that appear in patterns. The metafunction-style is used for the names of metafunctions. The non-terminal-style is for non-terminals and non-terminal-subscript-style is used for the portion after the underscore in non-terminal references.

The default-style is used for parenthesis, the dot in dotted lists, spaces, the separator words in the grammar, the "where" and "fresh" in side-conditions, and other places where the other parameters aren't used.

```
(label-font-size) → (and/c (between/c 1 255) integer?)
(label-font-size size) → void?
  size : (and/c (between/c 1 255) integer?)
(metafunction-font-size) → (and/c (between/c 1 255) integer?)
(metafunction-font-size size) → void?
  size : (and/c (between/c 1 255) integer?)
(default-font-size) → (and/c (between/c 1 255) integer?)
(default-font-size size) → void?
  size : (and/c (between/c 1 255) integer?)
```

These parameters control the various font sizes. The default-font-size is used for all of the font sizes except labels and metafunctions.

```
(reduction-relation-rule-separation)
 → (parameter/c (and/c integer? positive? exact?))
(reduction-relation-rule-separation sep) → void?
  sep : (parameter/c (and/c integer? positive? exact?))
```

Controls the amount of space between clauses in a reduction relation. Defaults to 4.

```
(curly-quotes-for-strings) → boolean?
(curly-quotes-for-strings on?) → void?
  on? : boolean?
```

Controls if the open and close quotes for strings are turned into “ and ” or are left as merely ".

Defaults to #t.

```
(current-text) → (-> string? text-style/c number? pict?)
(current-text proc) → void?
  proc : (-> string? text-style/c number? pict?)
```

This parameter's function is called whenever Redex typesets some part of a grammar, reduction relation, or metafunction. It defaults to slideshow's `text` function.

```
(set-arrow-pict!) → (-> symbol? (-> pict?) void?)
(set-arrow-pict! proc) → void?
  proc : (-> symbol? (-> pict?) void?)
```

This functions sets the pict for a given reduction-relation symbol. When typesetting a reduction relation that uses the symbol, the thunk will be invoked to get a pict to render it. The thunk may be invoked multiple times when rendering a single reduction relation.

```
(white-bracket-sizing)
 → (-> string? number? (values number? number? number? number?))
(white-bracket-sizing proc) → void?
  proc : (-> string? number? (values number? number? number? number?))
```

This parameter is used when typesetting metafunctions to determine how to create the ⟦⟧ characters. Rather than using those characters directory (since glyphs tend not to be available in PostScript fonts), they are created by combining two '[' characters or two ']' characters together.

The procedure accepts a string that is either `"["` or `"]"`, and returns four numbers. The first two numbers determine the offset (from the left and from the right respectively) for the second square bracket, and the second two two numbers determine the extra space added (to the left and to the right respectively).

The default value of the parameter is:

```
(λ (str size)
  (let ([inset-amt (floor/even (max 4 (* size 1/2)))])
    (cond
      [(equal? str "[")
       (values inset-amt
               0
               0
               (/ inset-amt 2))]
      [else
       (values 0
               inset-amt
               (/ inset-amt 2)
               0)])))
```

where `floor/even` returns the nearest even number below its argument. This means that for sizes 9, 10, and 11, `inset-amt` will be 4, and for 12, 13, 14, and 15, `inset-amt` will be 6.

Removing the pink background from PLT Redex rendered picts and ps files

When reduction rules, a metafunction, or a grammar contains unquoted Scheme code or side-conditions, they are rendered with a pink background as a guide to help find them and provide alternative typesettings for them. In general, a good goal for a PLT Redex program that you intend to typeset is to only include such things when they correspond to standard mathematical operations, and the Scheme code is an implementation of those operations.

To replace the pink code, use:

```
(with-unquote-rewriter proc expression)
```

It installs *proc* the current unqoute rewriter and evaluates expression. If that expression computes any picts, the unquote rewriter specified is used to remap them.

The *proc* should be a function of one argument. It receives a `lw` struct as an argument and should return another `lw` that contains a rewritten version of the code.

```
(with-atomic-rewriter name-symbol string-or-thunk-returning-pict expression)
```

This extends the current set of atomic-rewriters with one new one that rewrites the value of name-symbol to string-or-pict-returning-thunk (applied, in the case of a thunk), during the evaluation of expression.

name-symbol is expected to evaluate to a symbol. The value of string-or-thunk-returning-pict is used whever the symbol appears in a pattern.

```
(with-compound-rewriter name-symbol proc expression)
```

This extends the current set of compound-rewriters with one new one that rewrites the value of name-symbol via proc, during the evaluation of expression.

name-symbol is expected to evaluate to a symbol. The value of proc is called with a `(listof lw)`, and is expected to return a new `(listof (or/c lw? string? pict?))`, rewritten appropriately.

The list passed to the rewriter corresponds to the `lw` for the sequence that has name-symbol's value at its head.

The result list is constrained to have at most 2 adjacent non-`lw`s. That list is then transformed by adding `lw` structs for each of the non-`lw`s in the list (see the description of `lw` below for an explanation of logical-space):

- If there are two adjacent `lw`s, then the logical space between them is filled with whitespace.
- If there is a pair of `lw`s with just a single non-`lw` between them, a `lw` will be created (containing the non-`lw`) that uses all of the available logical space between the `lw`s.
- If there are two adjacent non-`lw`s between two `lw`s, the first non-`lw` is rendered right after the first `lw` with a logical space of zero, and the second is rendered right before the last `lw` also with a logical space of zero, and the logical space between the two `lw`s is absorbed by a new `lw` that renders using no actual space in the typeset version.

8.3 LW

```
(build-lw e line line-span column column-span) → lw?
  e : (or/c string?
            symbol?
            pict?
            (listof (or/c (symbols 'spring) lw?)))
  line : exact-positive-integer?
  line-span : exact-positive-integer?
  column : exact-positive-integer?
  column-span : exact-positive-integer?
(lw-e lw) → (or/c string?
                   symbol?
                   pict?
                   (listof (or/c (symbols 'spring) lw?)))
  lw : lw?
(lw-line lw) → exact-positive-integer?
  lw : lw?
(lw-line-span lw) → exact-positive-integer?
  lw : lw?
(lw-column lw) → exact-positive-integer?
  lw : lw?
(lw-column-span lw) → exact-positive-integer?
  lw : lw?
(lw? v) → boolean?
  v : any/c
lw
```

The lw data structure corresponds represents a pattern or a Scheme expression that is to be typeset. The functions listed above construct `lw` structs, select fields out of them, and recognize them. The `lw` binding can be used with `copy-struct`.

```
(to-lw arg)
```

This form turns its argument into lw structs that contain all of the spacing information just as it would appear when being used to typeset.

Each sub-expression corresponds to its own lw, and the element indicates what kind of subexpression it is. If the element is a list, then the lw corresponds to a parenthesized sequence, and the list contains a lw for the open paren, one lw for each component of the sequence and then a lw for the close parenthesis. In the case of a dotted list, there will also be a lw in the third-to-last position for the dot.

For example, this expression:

```
(a)
```

becomes this lw (assuming the above expression appears as the first thing in the file):

```
(build-lw (list (build-lw "(" 0 0 0 1)
                          (build-lw 'a 0 0 1 1)
                          (build-lw ")" 0 0 2 1))
                    0 0 0 3)
```

If there is some whitespace in the sequence, like this one:

```
(a b)
```

then there is no lw that corresponds to that whitespace; instead there is a logical gap between the lws.

```
(build-lw (list (build-lw "(" 0 0 0 1)
                (build-lw 'a 0 0 1 1)
                (build-lw 'b 0 0 3 1)
                (build-lw ")" 0 0 4 1))
          0 0 0 5)
```

In general, identifiers are represented with symbols and parenthesis are represented with strings and picts can be inserted to render arbitrary pictures.

The line, line-span, column, and column-span correspond to the logical spacing for the redex program, not the actual spacing that will be used when they are rendered. The logical spacing is only used when determining where to place typeset portions of the program. In the absense of any rewriters, these numbers correspond to the line and column numbers in the original program.

The line and column are absolute numbers from the beginning of the file containing the expression. The column number is not necessarily the column of the open parenthesis in a sequence – it is the leftmost column that is occupied by anything in the sequence. The line-span is the number of lines, and the column span is the number of columns on the last line (not the total width).

When there are multiple lines, lines are aligned based on the logical space (ie, the line/column & line-span/column-span) fields of the lws. As an example, if this is the original pattern:

```
(all good boys
     deserve fudge)
```

then the leftmost edges of the words "good" and "deserve" will be lined up underneath each other, but the relative positions of "boys" and "fudge" will be determined by the natural size of the words as they rendered in the appropriate font.

When 'spring appears in the list in the e field of a lw struct, then it absorbs all of the space around it. It is also used by to-lw when constructing the picts for unquoted strings. For example, this expression

```
,x
```

corresponds to these structs:

```
(build-lw (list (build-lw "" 1 0 9 0)
                'spring
                (build-lw x 1 0 10 1))
          1 0 9 2)
```

and the 'spring causes there to be no space between the empty string and the x in the typeset output.

```
(render-lw language/nts lw) → pict?
  language/nts : (or/c (listof symbol?) compiled-lang?)
  lw : lw?
```

Produces a pict that corresponds to the *lw* object argument, using *language/nts* to determine which of the identifiers in the *lw* argument are non-terminals.

This function sets dc-for-text-size. See also lw->pict.

```
(lw->pict language/ntw lw) → pict?
  language/ntw : (or/c (listof symbol?) compiled-lang?)
  lw : lw?
```

Produces a pict that corresponds to the *lw* object argument, using language/nts to determine which of the identifiers in the *lw* argument are non-terminals.

This does not set the dc-for-text-size parameter. See also render-lw.

```
(just-before stuff lw) → lw?
  stuff : (or/c pict? string? symbol?)
  lw : lw?
(just-after stuff lw) → lw?
  stuff : (or/c pict? string? symbol?)
  lw : lw?
```

These two helper functions build new lws whose contents are the first argument, and whose line and column are based on the second argument, making the new loc wrapper be either just before or just after that argument. The line-span and column-span of the new lw is always zero.

Index